TCS-TR-A-14-78

TCS Technical Report

Generating Sets of Permutations with Pattern Occurrence Counts Using πDDs

by

YUMA INOUE, TAKAHISA TODA AND SHIN-ICHI MINATO

Division of Computer Science

Report Series A

October 6, 2014

Hokkaido University
Graduate School of
Information Science and Technology

Email: yuma@ist.hokudai.ac.jp

Phone: +81-011-706-7681
Fax: +81-011-706-7683

Generating Sets of Permutations with Pattern Occurrence Counts Using πDDs

Yuma Inoue
Division of Computer Science
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo, Japan

Takahisa Toda
Graduate School of Information Systems
The University of Electro-Communications
Chofu, Tokyo

Shin-ichi Minato*
Division of Computer Science
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo, Japan

October 6, 2014

Abstract

A pattern *occurs* in a permutation if there is a subsequence of the permutation with the same relative order as the pattern. For mathematical analysis of permutation patterns, *strong Wilf-equivalence* has been defined as the equivalence between permutation patterns based on the number of occurrences of a pattern. In this paper, we present an algorithm for generating permutations of length n in which a pattern σ occurs exactly k times. Our approach is based on *permutation decision diagrams* (*πDDs*), which can represent and manipulate permutation sets compactly and efficiently. According to computational experiments, we gives a conjecture: all strongly Wilf-equivalent classes are trivial.

1 Introduction

A pattern σ *occurs* in a permutation π if there is a subsequence in π which is order isomorphic to σ. Two numerical sequences $a = a_1a_2 \dots a_n$ and $b = b_1b_2 \dots b_m$ are

*He also works for JST ERATO Minato Project.

order isomorphic if a and b have the same length and satisfy that $a_i < a_j$ if and only if $b_i < b_j$ for all i, j. Conversely, π *avoids* σ if σ does not occur in π.

The first research of permutation patterns in computer science is *stack sort*, in which we sort elements through a single stack [1]. Knuth showed that stack sortable permutations avoid the pattern 231. After permutation patterns were introduced, many researchers have actively studied the number of permutations which avoid a given pattern [2, 3]. Moreover, classifications of permutation patterns based on such numbers have also been examined. Two patterns σ_1 and σ_2 are *strongly Wilf-equivalent* if $|F_n^{(k)}(\sigma_1)| = |F_n^{(k)}(\sigma_2)|$ for all non-negative integers n and k, where $F_n^{(k)}(\sigma)$ denotes the set of permutations of length n in which σ occurs exactly k times [4]. Unfortunately, few results are known about strong Wilf-equivalence, as stated in the survey [5].

In order to break down the current situation, a computational method can be useful for researches of strong Wilf-equivalence. Namely, we computationally generate $F_n^{(k)}(\sigma)$ for a given pattern σ and confirm whether or not two patterns are strongly Wilf-equivalent by comparing $|F_n^{(k)}(\sigma)|$. This is not a formal proof, but we can obtain counter examples to disprove or candidates, which are probably strongly Wilf-equivalent classes. Unfortunately, $\sum_k |F_n^{(k)}(\sigma)|$ equals $n!$, which is huge even if n is small. Thus, it is important to consider an efficient generation algorithm.

The generation problem for pattern-avoiding permutations, i.e. $F_n^{(0)}(\sigma)$, has been studied actively. Theoretically efficient generation algorithms for some particular patterns were proposed [6, 7]. On the other hand, as an instance of practical results, *PermLab*, which is software enumerating and listing pattern-avoiding permutations, is developed by Albert [8]. However, as far as we know, no algorithm generating $F_n^{(k)}(\sigma)$ for any k has been proposed.

In this paper, we present a practically efficient algorithm for implicitly generating $F_n^{(k)}(\sigma)$. Our algorithm is based on a permutation decision diagram (πDD), which is a data structure for a compact representation of sets of permutations [9]. "Implicitly generating" means computing a compressed representation of permutations as πDD, not listing permutations explicitly. πDDs also have rich algebraic set operations such as union and intersection. The computation time of these operations depends on the size of πDDs and not on the number of permutations represented by πDDs. This is a key of our algorithm because almost previous works focused on polynomial-delay generation algorithms, whose time complexity depends on the number of permutations.

We carry out experiments to measure the performance of our algorithm. From the experimental results, we guess that the following conjecture holds: all strongly Wilf-equivalent classes are trivial, i.e., each class consists only of a permutation and its symmetric permutations. In addition, we also present some candidates of strongly Wilf-equivalent classes for generalized patterns, vincular patterns [10]. We hope that our algorithm could be a useful tool for promoting researches of strong Wilf-equivalence.

The rest of this paper is organized as follows. Section 2 introduces permutation patterns and πDDs. Section 3 presents our algorithm for generating $F_n^{(k)}(\sigma)$. Section 4 shows experimental results by a program based on our algorithm. Section 5 concludes this paper.

2 Preliminaries

2.1 Permutations

A permutation of length n is a bijection from $\{1, 2, ..., n\}$ to itself, and hereafter, we call it an n-permutation for brevity. Let π be an n-permutation. We write a permutation in the one-line form as $\pi = \pi(1)\pi(2)\ldots\pi(n)$, and denote $\pi_i = \pi(i)$. For example, $\pi = 3421$ is a 4-permutation and $\pi_3 = 2$. We define S_n as the set of all n-permutations.

Multiplication on permutations x and y is defined as $x \cdot y = y_{x_1} y_{x_2} \ldots y_{x_n}$, which is y after applying x. In other words, if we consider a one-line form of a permutation as a numerical sequence, a multiplication $x \cdot y$ is a rearrangement of y according to the order of x. For example, let π be a 5-permutation, then $54321 \cdot \pi = \pi_5\pi_4\pi_3\pi_2\pi_1$ is the reverse of π.

A *transposition* is a permutation by which exactly two elements are swapped. More precisely, a transposition $\tau_{i,j}$ is a permutation such that $\tau_{i,j}(i) = j$, $\tau_{i,j}(j) = i$, and $\tau_{i,j}(k) = k$ for all other numbers k. Any n-permutation can be uniquely represented as the product of at most $n-1$ transpositions using a straight selection sorting algorithm. This algorithm repeats to swap the value k and the kth element, where k runs from n to 1. For example, we consider a decomposition of the permutation 43152 into a product of transpositions. The 5th element of 43152 is 2, hence we exchange 5 and 2, and obtain $43152 = 43125 \cdot \tau_{2,5}$. Since the 4th element of 43125 is 2, we then obtain $43152 = 23145 \cdot \tau_{2,4} \cdot \tau_{2,5}$. Repeating this process, we finally obtain $43152 = \tau_{1,2} \cdot \tau_{1,3} \cdot \tau_{2,4} \cdot \tau_{2,5}$.

2.2 Permutation Patterns

A permutation σ *occurs* in a permutation π if there is at least one subsequence in π which is order isomorphic to σ, where such a subsequence does not have to be consecutive in π. In other words, let l be the length of σ, then σ occurs in π if there are indices $1 \le i_1 < i_2 < \ldots < i_l \le n$ such that $\pi_{i_x} < \pi_{i_y}$ if and only if $\sigma_x < \sigma_y$, for all pairs of x and y. Such σ is called a *pattern*. For example, the pattern 312 occurs in the permutation 4213 because 423 and 413 are order isomorphic to the pattern 312. On the other hand, a permutation π *avoids* a pattern σ if there is no occurrence of σ in π.

The *pattern occurrence count* of σ in π is the number of distinct subsequences in the permutation π which are order isomorphic to the pattern σ. Hereafter, $F_n^{(k)}(\sigma)$

denotes the set of n-permutations in which σ occurs exactly k times. Two patterns σ_1 and σ_2 are *Wilf-equivalent* if they have the same number of n-permutations which avoid the patterns, i.e., $|F_n^{(0)}(\sigma_1)| = |F_n^{(0)}(\sigma_2)|$ holds for all positive integers n. More generally, two patterns σ_1 and σ_2 are *strongly Wilf-equivalent* if $|F_n^{(k)}(\sigma_1)| = |F_n^{(k)}(\sigma_2)|$ holds for all non-negative integers n and k.

The problem considered in this paper can be stated as follows: given a positive integer n and a pattern σ, generate $F_n^{(k)}(\sigma)$ for all non-negative integers k.

2.3 Permutation Decision Diagrams

A permutation decision diagram (πDD) is a data structure representing a set of permutations canonically [9]. πDDs have efficient set operations for permutation sets. Our algorithm is based on the compact representation and rich set operations of πDDs. πDDs are derived from *zero-suppressed binary decision diagrams* (*ZDDs*) [11], which are decision diagrams for sets of combinations. πDDs consist of five components: internal nodes with a transposition label, 0-edges, 1-edges, the 0-sink, and the 1-sink. An example of a πDD is shown in Fig. 1.

A πDD forms a binary decision diagram: each internal node has exactly a 0-edge and a 1-edge. Each path on a πDD represents a permutation: if a 1-edge originates from $\tau_{x,y}$, the decomposition of the permutation contains $\tau_{x,y}$, while a 0-edge from $\tau_{x,y}$ means that the decomposition excludes $\tau_{x,y}$. If a path reaches the 1-sink, the permutation corresponding to the path is in the set represented by the πDD. On the other hand, if a path reaches the 0-sink, the permutation is not in the set. For a node N, we call the subgraph pointed to by its 0-edge and 1-edge the *left subgraph* and the *right subgraph* of N, respectively.

A πDD has a compact and canonical form if we fix the order of transpositions and apply the following two reduction rules:

(1) Merging rule: share all nodes having the same labels, left subgraphs, and right subgraphs.
(2) Deletion rule: delete all nodes whose 1-edge points directly to the 0-sink.

These rules are illustrated in Fig. 2. We introduce the order of transpositions $<$ so that $\tau_{x_1,y_1} < \tau_{x_2,y_2}$ is true if $y_1 > y_2$ holds or $y_1 = y_2$ and $x_1 < x_2$ holds.

The *size* of a πDD (i.e. the number of nodes in a πDD) can grow exponentially with respect to the length of permutations. In many practical cases, though, a πDD demonstrates high compression ratio.

In addition, πDDs support efficient set operations such as union and intersection. Table 1 shows the πDD operations used in this paper. The computation time of πDD operations depends on only the size of πDDs, not on the cardinality of the sets represented by the πDDs. Note that *Cartesian product* of πDDs differs from usual Cartesian product of sets: Cartesian product of two πDDs P and Q means the set of all products for all pairs of permutations in P and Q.

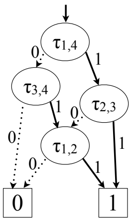


Figure 1: The πDD for a permutation set $\{4132, 2143, 3412\}$.

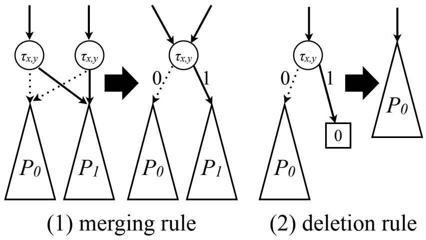


Figure 2: Two reduction rules on πDDs.

2.4 Multiset of Permutations

We introduce some definitions and notations for multisets of permutations. Let $\mathbf{P} = \langle P, f \rangle$ be a multiset over P, where P is a permutation set and $f : P \to \mathbb{N}$ is a function. Here, $f(\pi)$ is a multiplicity of a permutation π and we use the notation $\mathbf{P}(\pi)$ instead of $f(\pi)$ for convenience. We define $\mathbf{P} \uplus \mathbf{Q}$ as the *multiset sum* of two multisets $\mathbf{P}$ and $\mathbf{Q}$, where $(\mathbf{P} \uplus \mathbf{Q})(\pi) = \mathbf{P}(\pi) + \mathbf{Q}(\pi)$ holds for all permutations π. *Scalar multiplication* of an integer k and a multiset $\mathbf{P}$ is defined by $k \cdot \mathbf{P} = \langle P, k \cdot f(\pi) \rangle$. Cartesian product of two multisets $\mathbf{P} = \langle P, f \rangle$ and $\mathbf{Q} = \langle Q, g \rangle$ is defined by $\mathbf{P} * \mathbf{Q} = \langle P * Q, h(\pi \cdot \pi') = f(\pi) \cdot g(\pi') \rangle$, where $P * Q$ means Cartesian product of permutation sets like that of πDDs.

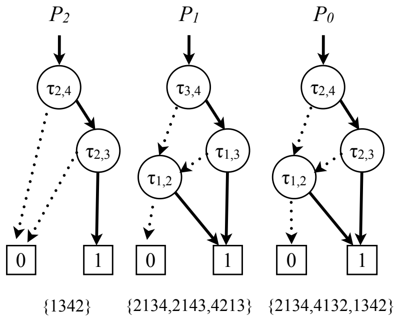
A set of permutations weighted by pattern occurrence counts can be represented by a multiset. Given a positive integer n and a pattern σ, $\mathbf{P}_{n,\sigma} = \langle S_n, f_\sigma \rangle$ denotes the set of n-permutations weighted by the pattern occurrence counts of σ, i.e., $f_\sigma(\pi)$ equals the pattern occurrence counts of σ in π. Our algorithm which will be described in Sect. 3 generates $\mathbf{P}_{n,\sigma}$ at first, and next computes $F_n^{(k)}(\sigma) = \{\pi \mid \mathbf{P}_{n,\sigma}(\pi) = k\}$ from $\mathbf{P}_{n,\sigma}$.

2.5 πDD Vector

In this paper, we want to handle multisets of permutations. But πDDs cannot represent multisets. To overcome this problem, we use a *πDD vector*, proposed in [12]. A πDD vector represents a multiset of permutations based on a binary

Table 1: πDD operations on two πDDs P and Q.

$P.Top$	return the transposition $\tau_{x,y}$ by which the root node is labeled.
$P \cap Q$	return intersection $\{\pi \mid \pi \in P$ and $\pi \in Q\}$.
$P \cup Q$	return union $\{\pi \mid \pi \in P$ or $\pi \in Q\}$.
$P \setminus Q$	return difference $\{\pi \mid \pi \in P$ and $\pi \notin Q\}$.
$P \cdot \tau(x, y)$	return swap $\{\pi \cdot \tau_{x,y} \mid \pi \in P\}$.
$P * Q$	return Cartesian product $\{\alpha \cdot \beta \mid \alpha \in P$ and $\beta \in Q\}$.



Permutation	(P_2, P_1, P_0)	Multiplicity
1342	(1,0,1)	5
2134	(0,1,1)	3
2143	(0,1,0)	2
4213	(0,1,0)	2
4132	(0,0,1)	1

Figure 3: An example of a πDD vector.

Table 2: πDD operations on πDD vectors $\vec{P}$ and $\vec{Q}$.

$\vec{P} \uplus \vec{Q}$	Return multiset sum $\vec{R}$ such that $\vec{R}(\pi) = \vec{P}(\pi) + \vec{Q}(\pi)$ holds.
$\vec{P} \cdot \tau(x, y)$	Return swap $\vec{P'} = (P_0 \cdot \tau(x, y), P_1 \cdot \tau(x, y), \ldots, P_m \cdot \tau(x, y))$.
$\vec{P}.numberof(\pi)$	Return $\vec{P}(\pi)$.

representation using multiple πDDs. A πDD vector consists of an array of πDDs. Let M be the maximum multiplicity of a permutation in a given multiset. We denote a πDD vector by $\vec{P} = (P_0, P_1, \ldots, P_m)$, where each P_i is a πDD and $m = \lfloor \log M \rfloor$. Let $\vec{P}(\pi)$ be the multiplicity of π in $\vec{P}$. If a permutation π is in P_i, the ith bit of the binary representation of $\vec{P}(\pi)$ is 1, and otherwise 0. In other words, $\vec{P}(\pi) = \sum_{i=0}^{m} 2^i \cdot [\pi \in P_i]$ holds, where $[x]$ equals 1 if x is true, and otherwise 0. Figure 3 shows an example of a πDD vector. πDD vectors have multiset operations. Table 2 shows some πDD vector operations which were proposed in [12].

3 Main Results

Our algorithm is divided into two parts: generation of $\mathbf{P}_{n,\sigma}$ and generation of $F_n^{(k)}(\sigma)$ from $\mathbf{P}_{n,\sigma}$. The first part is described in Sects. 3.1 and 3.2. The second part is described in Sect. 3.3.

3.1 Existing Method and Its Extension

We proposed a generation algorithm for non-weighted occurrence using πDD in [13]. In [13], in order to generate pattern-avoiding permutations, we generate S_n and $C_n(\sigma)$, which is the set of all n-permutations in which a given patten occurs at least once, and calculate the set difference $S_n \setminus C_n(\sigma)$. Thus, in order to construct the πDD vector for $\mathbf{P}_{n,\sigma}$, we want to extend the generation algorithm for $C_n(\sigma)$ to handle multiplicities as the pattern occurrence counts of σ.

Algorithm 1 Scalar multiplication of a non-negative integer $k = (k_l, k_{l-1}, \ldots, k_0)_2$ and a πDD vector $\vec{P} = (P_0, P_1, \ldots, P_m)$.

```
πDD vector R⃗ ← (∅)
for i = 0 to l do
  if k_i = 1 then
    R⃗ ← R⃗ ⊎ P⃗
  end if
  P⃗ ← (∅, P_0, P_1, ..., P_{m+i})
end for
return R⃗
```

The algorithm for generating $C_n(\sigma)$ in [13] is sketched as follows. Let l be the length of σ. At first, we generate all numerical sequences satisfying the following conditions: it is order isomorphic to σ and all the elements in it are less than or equal to n. The number of such numerical sequences is $\binom{n}{l}$. Next, we generate all n-permutations such that the above numerical sequences appears as their subsequence. In order to do this, it is sufficient to embed the numerical sequences in sequences of length n without changing their order. Here, we assign other numbers with other positions in any order. Since all order isomorphic sequences are embedded in all possible positions, this process generates $C_n(\sigma)$.

In [13], we used Cartesian product operations of πDDs to realize this process. As described in Sect. 2.1, a multiplication of permutations can be considered as a rearrangement. Since the above embedding can be considered as a rearrangement, we can generate $C_n(\sigma)$ by calculating the Cartesian product of the two πDDs: the permutations which represent all possible positions and the permutations which represent all order isomorphic numerical sequences. These correspond to the πDDs $\mathbb{C}$ and $\mathbb{B} * \mathbb{A}$ in [13], respectively.

Here, note that some embeddings cause duplications. For example, both 413 and 524, which are order isomorphic to 312, are embedded in 52413. In other words, 312 occurs (at least) 2 times in 52413. The number of such duplications is equal to pattern occurrence counts of a given pattern in a permutation. Since the existing method uses πDD, such duplications are not counted. But by using the Cartesian product of πDD vectors instead of πDDs, we can generate the multiset of permutations whose multiplicity represents the pattern occurrence count, i.e. $\mathbf{P}_{n,\sigma}$. Unfortunately, however, Cartesian product of πDD vectors have not ever been proposed. We introduce the operation in the next subsection.

3.2 Cartesian Product of πDD Vector

Cartesian product between two πDDs P and Q was defined as follows ([9]):

$$P * Q = (P * Q^l) \cup ((P * Q^r) \cdot \tau(x, y)),$$

where Q^l and Q^r are the left and the right subgraphs of the root node of Q, respectively, and $\tau_{x,y}$ is the transposition associated with the root node of Q.

We extend Cartesian product to πDD vectors as follows:

$$\vec{P} * \vec{Q} = (\vec{P} * \vec{Q}^L) \uplus ((\vec{P} * \vec{Q}^R) \cdot \tau(x, y)),$$

where multiset sum $\uplus$ and swap $\tau(x, y)$ are πDD vector operations which were already proposed in [12] as shown in Table 2. However, some Q_i in the array of $\vec{Q}$ can have distinct transpositions in their root nodes. We need to define $\vec{Q}^L$ and $\vec{Q}^R$, and the pair (x, y) for $\tau(x, y)$ appropriately.

We use the pair (x_s, y_s) such that τ_{x_s,y_s} is the smallest transposition in the root nodes of $Q_1, Q_2, \ldots, Q_m$, where the order of transpositions is introduced in Sect. 2.2. For example, for the πDD vector illustrated in Fig. 3, (x_s, y_s) is $(2, 4)$ because the root nodes of P_0, P_1, and P_2 are $\tau_{2,4}$, $\tau_{3,4}$, and $\tau_{2,4}$, respectively, and the smallest transposition is $\tau_{2,4}$. Here, we define $\vec{Q}^L$ and $\vec{Q}^R$ as follows:

$$\vec{Q}^L = (Q_0^L, Q_1^L, \ldots Q_m^L), \text{where } Q_i^L = \begin{cases} Q_i \text{ if } Q_i.Top \neq \tau_{x_s,y_s}, \\ Q_i^l \text{ otherwise.} \end{cases}$$

$$\vec{Q}^R = (Q_0^R, Q_1^R, \ldots Q_m^R), \text{where } Q_i^R = \begin{cases} \emptyset \text{ if } Q_i.Top \neq \tau_{x_s,y_s}, \\ Q_i^r \text{ otherwise.} \end{cases}$$

It works because Q_i whose root node is not labeled by $\tau_{x,y}$ is equivalent to the πDD whose root node has the label $\tau_{x,y}$ and Q_i as the left subgraph, the 0-sink as the right subgraph due to the deletion rule in Sect. 2.2.

We also need to define the recursion basis. The recursion basis means that for all i, Q_i consists only of a sink node. If every root node is a 0-sink, the Cartesian product is an empty set. Otherwise, $\vec{Q}(12\ldots n) = k$ and $\vec{Q}(\pi) = 0$ for the other permutations π hold, and then $\vec{P} * \vec{Q}$ equals the scalar multiplication $k \cdot \vec{P}$. Algorithm 1 represents an algorithm for a scalar multiplication of a πDD vector based on the multiset sum of $2^i \cdot \vec{P} = (\underbrace{\emptyset, \ldots, \emptyset}_{i}, P_0, \ldots, P_m)$.

Cartesian product for πDD vectors is described in Algorithm 2. In conclusion, by using this Cartesian product of πDD vectors, we can implicitly generate $\mathbf{P}_{n,\sigma}$.

3.3 Generation of $F_n^{(k)}(\sigma)$

We propose an algorithm generating $\mathbf{P}_{n,\sigma}$. However, in order to computationally check whether or not strong Wilf-equivalence between σ_1 and σ_2 holds, we must calculate the cardinalities of $F_n^{(k)}(\sigma_1)$ and $F_n^{(k)}(\sigma_2)$ for $0 \leq k \leq M$, where M denotes the maximum number of occurrences of a given pattern. Note that $M \leq \binom{n}{l}$ holds, where l is the length of the pattern. In this subsection, we propose the algorithm constructing the πDDs for $F_n^{(k)}(\sigma)$ for $0 \leq k \leq M$ from the πDD vector for $\mathbf{P}_{n,\sigma}$.

Algorithm 2 Cartesian product of two πDD vectors $\vec{P} = (P_0, P_1, \ldots, P_{m'})$ and $\vec{Q} = (Q_0, Q_1, \ldots, Q_m)$.

{We suppose the 0-sink with $\tau_{0,0}$ and the 1-sink with $\tau_{1,1}$ for convenience.}
transposition $t \leftarrow \tau_{0,0}$
for $i = 0$ to m **do**
  if $Q_i.Top < t$ **then**
    $t \leftarrow Q_i.Top$
  end if
end for

if $t = \tau_{0,0}$ **then**
  return $(\emptyset)$
else if $t = \tau_{1,1}$ **then**
  return $\vec{Q}.numberof(12\ldots n) \cdot \vec{P}$
else
  for $i = 0$ to m **do**
    if $Q_i.Top = t$ **then**
      $Q_i^L \leftarrow Q_i^l$, $Q_i^R \leftarrow Q_i^r$
    else
      $Q_i^L \leftarrow Q_i$, $Q_i^R \leftarrow \emptyset$
    end if
  end for
  return $(\vec{P} * \vec{Q}^L) \uplus ((\vec{P} * \vec{Q}^R) \cdot t)$
end if

Let m be $\lfloor \log M \rfloor$. If k is fixed, it is easy to calculate $F_n^{(k)}(\sigma)$ from the πDD vector $\vec{P} = (P_0, P_1, \ldots, P_m)$ as follows:

$$F_n^{(k)}(\sigma) = \bigcap_{0 \le i \le m} \{P_i \mid k_i = 1\} \setminus \bigcup_{0 \le i \le m} \{P_i \mid k_i = 0\},$$

where k_i means ith bit of the binary representation of k. This algorithm costs $O(m)$ πDD operations. Hence, computation of $F_n^{(k)}(\sigma)$ for $0 \le k \le M$ costs $O(mM)$ πDD operations.

We improve the number of πDD operations from $O(mM)$ to $O(M)$. Let W_k be the πDD for the set of permutations whose multiplicity is k in the given πDD vector $\vec{P}$. We introduce *don't care* * to the binary representation of integers. Here, $W_{(*\ldots*0k_i\ldots k_0)_2} \cup W_{(*\ldots*1k_i\ldots k_0)_2} = W_{(*\ldots**k_i\ldots k_0)_2}$ and $W_{(*\ldots*0k_i\ldots k_0)_2} \cap W_{(*\ldots*1k_i\ldots k_0)_2} = \emptyset$ hold. Hence,

$$\begin{aligned} W_{(*\ldots*1k_i\ldots k_0)_2} &= W_{(*\ldots**k_i\ldots k_0)_2} \cap P_{i+1}, \\ W_{(*\ldots*0k_i\ldots k_0)_2} &= W_{(*\ldots**k_i\ldots k_0)_2} \setminus W_{(*\ldots*1k_i\ldots k_0)_2}. \end{aligned}$$

Therefore, we can calculate W_k for $0 \le k \le M$ from $W_{(*\ldots*)_2} = S_n$ by repeating calculations of the above recursions. An algorithm generating S_n is also shown

Algorithm 3 Generate W_k for all $0 \le k \le M$ from $\vec{P} = (P_0, P_1, \ldots, P_m)$.

```
W_0 ← πDD for S_n
for i = 0 to m do
  for bin = 2^i to 2^{i+1} − 1 do
    W_bin ← W_{bin−2^i} ∩ P_i
    W_{bin−2^i} ← W_{bin−2^i} \ W_bin
  end for
end for
```

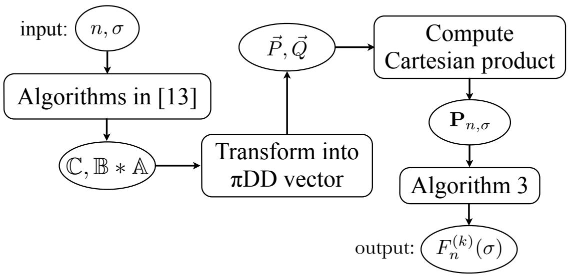


Figure 4: The summary of our algorithm.

in [13]. The number of valid binary representations whose prefix can be consecutive "don't care" symbols is $2^0 + 2^1 + \ldots + 2^m = 2^{m+1} - 1 = O(M)$. For each calculation of W_k, we use only one πDD operation. Therefore, we can generate $F_n^{(k)}(\sigma)$ for $0 \le k \le M$ by $O(M)$ πDD operations based on the recursions. Algorithm 3 shows a pseudo code of this algorithm. This algorithm temporarily uses $W_{(0\ldots 0k_i\ldots k_0)_2}$ to represent $W_{(*\ldots *k_i\ldots k_0)_2}$.

3.4 Summary of Our Algorithm and Its Extension

Our algorithm can be summarized as follows. First, we construct the πDDs for $\mathbb{C}$ and $\mathbb{B} * \mathbb{A}$ by the algorithms described in [13]. Second, we transform these πDDs into the πDD vectors, that is, $\vec{P} = (\mathbb{C})$ and $\vec{Q} = (\mathbb{B} * \mathbb{A})$. Third, we calculate the Cartesian product $\vec{P} * \vec{Q}$, which is the πDD vector for $\mathbf{P}_{n,\sigma}$. Finally, we calculate the πDDs for $F_n^{(k)}(\sigma)$ from the πDD vector for $\mathbf{P}_{n,\sigma}$ using Algorithm 3. This process is illustrated in Fig. 4.

In addition, we can easily extend our algorithm for other generalized patterns such as *vincular patterns* [10] and *bivincular patterns* [14]. In [13], in order to extend the algorithm to deal with such patterns, we only change $\mathbb{A}$ and $\mathbb{C}$ to $\mathbb{A}'$ and $\mathbb{C}'$, respectively. Thus, by using $\mathbb{A}'$ and $\mathbb{C}'$ in [13], we can generate $F_n^{(k)}(\sigma)$ for such patterns.

Table 3: Computation time (sec) for generating $F_n^{(k)}(\sigma)$.

n		πDD Method l				Naive Method l			
		2	3	4	5	2	3	4	5
9	best	0.076	0.212	0.232	0.068	0.316	0.676	1.012	0.968
	average	0.078	0.295	0.274	0.087	0.322	0.714	1.088	1.037
	worst	0.080	0.356	0.308	0.120	0.328	0.740	1.164	1.128
10	best	0.276	1.912	2.820	0.956	3.792	10.301	20.141	25.522
	average	0.310	2.604	3.390	1.232	3.900	10.891	21.489	27.314
	worst	0.344	3.288	3.908	1.524	4.008	11.393	23.013	29.482
11	best	1.436	14.165	30.190	11.673	51.595	169.331	467.185	750.307
	average	1.642	18.943	38.814	15.683	53.673	179.724	488.088	774.683
	worst	1.848	24.098	48.331	19.809	55.752	188.248	509.804	806.186
12	best	6.316	96.114	365.275	144.141	—	—	—	—
	average	7.064	136.343	494.926	231.141	—	—	—	—
	worst	7.812	180.711	655.829	301.135	—	—	—	—

4 Experimental Results

We implemented our algorithms in C++ and carried out experiments on a 3.20 GHz CPU machine with 64 GB memory. We compared the performance of our algorithm to that of a naive method. The naive method generates all n-permutations and, for each n-permutation, enumerates the pattern occurrence counts of σ by checking the order isomorphism between all k-subsequences and σ.

Table 3 shows computation time for the entire process of generating $F_n^{(k)}(\sigma)$. The table represents the best, the worst, and the average computation time over all patterns with length l = 2,3,4, and 5, respectively. Note that computation time of our πDD method only include the constructions of πDDs, meaning that we do not output permutations explicitly. The naive method only counts $|F_n^{(k)}(\sigma)|$, and not explicitly outputs permutations too. For $n = 10$ and $l = 5$, our algorithm is about 20 times faster than the naive method. Our algorithm takes the maximum computation time when l is near $n/3$, while the naive method takes the maximum time when l is near $n/2$. We consider that this is because the difference of the parameters dominating computation time: the computation time of a πDD depends on the size of the πDD while that of the naive method depends on the number of subsequences, i.e. $\binom{n}{l}$. In practical cases, the size of a πDD tends to become small when the set of permutations is sparse or very dense. When σ is a long pattern, $F_n^{(k)}(\sigma)$ tends to be dense for small k and to be sparse for large k. That is, the longer the length of a pattern is, the smaller the πDDs for the pattern tend to be.

Table 4 describes memory consumption and the size of πDDs for generating $F_n^{(k)}(\sigma)$. We do not describe memory consumption of the naive method because the naive method does not store the permutations and uses a small memory. Tables 3 and 4 confirm the computation time is proportional to the size of πDDs. Memory consumption of our algorithm grows exponentially with respect to n. We cannot calculate $F_{13}^{(k)}(\sigma)$ because of memory shortage. However the total size of πDDs for $F_n^{(k)}(\sigma)$ is smaller than the number of all n-permutations $n!$. This shows that πDDs achieve compact representations of $F_n^{(k)}(\sigma)$.

Table 4: Memory consumption and the size of πDDs for generating $F_n^{(k)}(\sigma)$.

n		memory consumption (kB) l				$\sum_{k=0}^{M}\{$the size of πDDs for $F_n^{(k)}(\sigma)\}$ l			
		2	3	4	5	2	3	4	5
9	best	38340	41528	41392	38604	42496	139315	86321	30843
	average	38736	42542	42372	39076				
	worst	39132	43156	42940	39660				
10	best	42480	152316	268348	78532	167368	962945	726510	262833
	average	43272	212905	275364	141172				
	worst	44064	275604	279712	154728				
11	best	152328	1032672	2064776	1031504	658823	6684948	6807355	2548799
	average	152484	1092093	2142647	1067605				
	worst	152640	1160728	2224848	1128760				
12	best	520216	4209304	16461940	8263752	2585682	45156225	69564400	27639470
	average	527320	6863648	18459651	9723839				
	worst	534424	8232056	32355008	16375948				

Table 5: Candidates of strongly Wilf-equivalent classes for vincular patterns.

124-3,134-2,2-134,2-431,3-124,3-421,421-3,431-2	132-4,1-324,142-3,2-314,3-241,413-2,423-1,4-231
2-14-3,2-41-3,3-14-2,3-41-2	1342,1432,2341,2431,3124,3214,4123,4213
13-24,24-13,31-42,42-31	1-423,231-4,241-3,2-413,314-2,3-142,324-1,4-132

In the experiments, we could not find non-trivial classes for $2 \leq l \leq 8$. In other words, we computationally prove that there are only trivial strongly Wilf-equivalent classes for $2 \leq l \leq 8$. Note that it is trivial that a pattern and its symmetric permutations, i.e. its reverse, complement, and inverse, are always strongly Wilf-equivalent. Since we carried out experiments only for $2 \leq n \leq 12$, many non-trivial candidates are found for $l = 9$. But the experiments for $n > 12$ may confirm there are only trivial classes for $l = 9$. In fact, in the case $l = 8$, the results for $n < 12$ has many non-trivial candidates while the result for $n = 12$ confirms there are only trivial classes. Therefore, we propose a conjecture: all strongly Wilf-equivalent classes are trivial for all l. As far as we know, this conjecture have never been proven.

We also carried out experiments for vincular pattens [10] for $2 \leq n \leq 12$. Vincular patterns are generalized patterns with adjacent constraints: if there is no hyphen (-) between σ_i and σ_{i+1}, the corresponding elements in a permutation must be adjacent. Table 5 provides non-trivial candidates for the vincular pattens with length 4. The classes in the left column have been proved in [4]. On the other hand, the classes in the right column have never been proven as far as we know. Thus, we propose the following conjectures: the vincular pattens (1342, 1432), (1-423, 2-413), and (132-4, 142-3) are strongly Wilf-equivalent respectively.

5 Conclusion

In this paper, we proposed an algorithm implicitly generating $\mathbf{P}_{n,\sigma}$ for given n and σ using πDD vectors. Furthermore, we provided an algorithm computing the πDDs representing $F_n^{(k)}(\sigma)$ for each k from the πDD vector for $\mathbf{P}_{n,\sigma}$. Experimental results present that our algorithm is practically faster than a naive method, and suggest conjectures about strongly Wilf-equivalence.

Future work is to improve the performance of our algorithms, especially memory consumption which is a bottleneck of our algorithm. Experiments for larger n and k, multiple patterns, and other general patterns are also interesting for us. We wish that someone will prove our conjectures.

References

[1] Knuth, D.E.: Fundamental Algorithms. Third edn. Volume 1 of The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts (1997)

[2] Bóna, M.: Exact enumeration of 1342-avoiding permutations: A close link with labeled trees and planar maps. Journal of Combinatorial Theory, Series A **80**(2) (1997) 257–272

[3] Marinov, D., Radoičić, R.: Counting 1324-avoiding permutations. The Electronic Journal of Combinatorics **9**(2) (2003)

[4] Kasraoui, A.: New Wilf-equivalence results for vincular patterns. European Journal of Combinatorics **34**(2) (2013) 322–337

[5] Steingrímsson, E.: Some open problems on permutation patterns. London Mathematical Society Lecture Note Series (2013) 239–263

[6] Wilf, H.: The patterns of permutations. Discrete Math. **257** (2002) 575–583

[7] Dukes, W., Flanagan, M., Mansour, T., Vajnovszki, V.: Combinatorial gray codes for classes of pattern avoiding permutations. Theoretical Computer Science **396**(13) (2008) 35–49

[8] Albert, M.: Permlab: Software for permutation patterns. http://www.cs.otago.ac.nz/staffpriv/malbert/permlab.php (2012)

[9] Minato, S.: πDDs: A new decision diagram for efficient problem solving in permutation space. Proc. of 14th International Conference on Theory and Applications of Satisfiability Testing (2011) 90–104

[10] Babson, E., Steingrímsson, E.: Generalized permutation patterns and a classification of the Mahonian statistics. Séminaire Lotharingien de Combinatoire **44** (2000)

[11] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. Proc. of 30th ACM/IEEE Design Automation Conf.(DAC-93) (1993) 272–277

[12] Kawahara, J., Saitoh, T., Yoshinaka, R., Minato, S.: Counting primitive sorting networks by πDDs. Technical Report TCS-TR-A-11-54, Division of computer science, Hokkaido University (2011)

[13] Inoue, Y., Toda, T., Minato, S.: Implicit generation of pattern-avoiding permutations based on πDD. Technical Report TCS-TR-A-13-67, Division of computer science, Hokkaido University (2013)

[14] Bousquet-Mélou, M., Claesson, A., Dukes, M., Kitaev, S.: (2+2)-free posets, ascent sequences and pattern avoiding permutations. J. Comb. Theory Ser. A **117**(7) (Oct 2010) 884–909